CST8277 (22F) Lab Exercise #3

Notes for Lab Exercise #3

Theme for Lab Exercise #3

Application Performance Monitoring (APM) is the monitoring and management of performance of software applications. APM strives to detect and diagnose complex application problems, typically to meet an expected QoS (Quality-of-Service) – which may be a ***contractual commitment*** to end-users, i.e. the organization may lose money (or customers) if the QoS is not met. After completing this lab, you will have achieved the following:

1) Use Java Mission Control (JMC) and Java Flight Recorder (JFR) APM tools
2) Added the Java Mission Control (JMC) tool to Eclipse
3) Observed captured Java Flight Recorder (JFR) data

Java Flight Recorder (JFR)

The job of the Java Flight Recorder (JFR) tool is to ***profile*** a running Java program. When we say 'profiling' in the context of software, we mean that the *characteristics* of the program are captured and analyzed. Typical interesting run-time characteristics are:

- The memory used by a program: a.k.a space complexity
- The execution time of a program: a.k.a time complexity
- The *call graph* of a program: for each class, which other classes and methods are called and from those methods… and so on. Additionally, depending on the capability of the profiling system, show a running count of how many times each method is called.

Space Complexity

The simplest space characteristic we may wish to know is if a program has enough memory to operate. This is very important in environments restricted by RAM such as a 'smart' thermostat which typically do not have very much RAM (nor can the user add more later). We may wish to determine if a program has a memory leak. A program may seem to operate normally, but due to a bug allocates an unusually large amount of memory (without deallocating the memory after using it), more than what is available and crashes. ☹

Alternatively, the program may have a 'slow' memory leak: something relatively small but allocated too many times, resulting in the same outcome. 'Slow' memory leaks are particularly problematic because you may not find out about them until AFTER your program is running in production for hours, days, or even weeks. 😋

Time Complexity

The simplest time characteristic we may wish to know is if a program can complete a task within a certain time. This is very important in environments where the responsiveness of a program is important, for example, an industrial device that must decide to turn on or off a switch.

The program may be responsive under normal circumstances but under some conditions may not perform quickly enough. Either a specific method takes too long or – similar to the 'slow' leak issue above – a method may take a reasonable amount of time, but the program executes that method too many times.

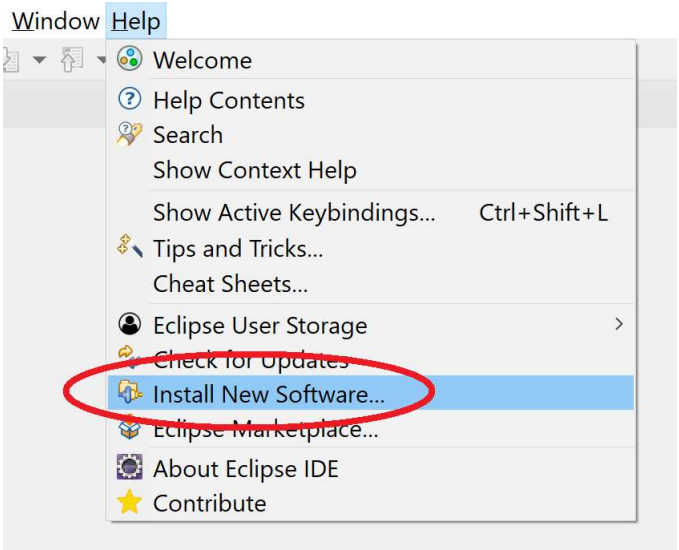
Installing Java Mission Control (JMC)

Java Mission Control (JMC) is a set of open source tools specific to the Java Virtual Machine. These tools are typically used to find problems and help optimize Java programs running in production. JMC supports OpenJDK 11 (and above): an earlier version was specific to Oracle's JDKs and was only available for paying customers. JMC primarily consists of:

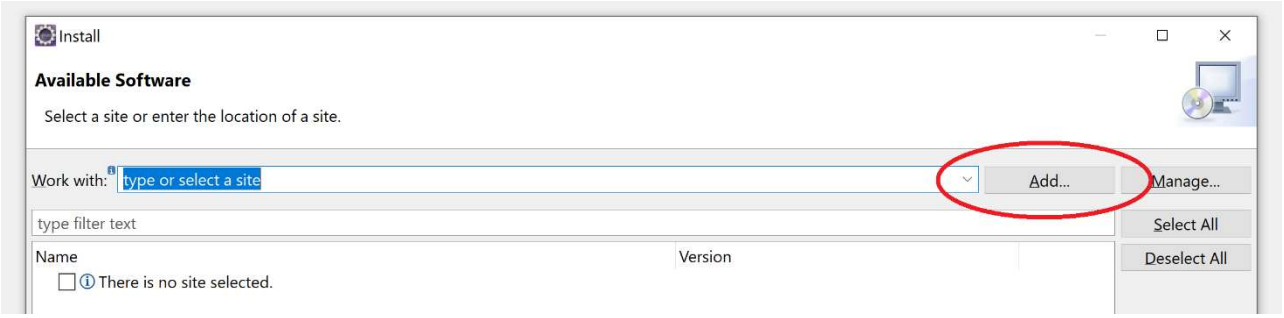
- Java Flight Recorder (JFR): general analyzer and visualizations
- Java Management Extension (JMX) Console: 'live' connect to JVMs
- Additional optional plugins: analyze thread dumps; JavaFX phases and input events, etc.

JMC can installed and run by itself or can be installed as an Eclipse plugin. Here, we will install JMC as an Eclipse plugin. First, please download from Brightspace ('Activities' -> 'Assignments' ->
Lab 3 - Java Mission Control and Java Flight Recorder) the archive file org.openjdk.jmc.updatesite.ide-8.1.1.zip and save it on your local hard drive.

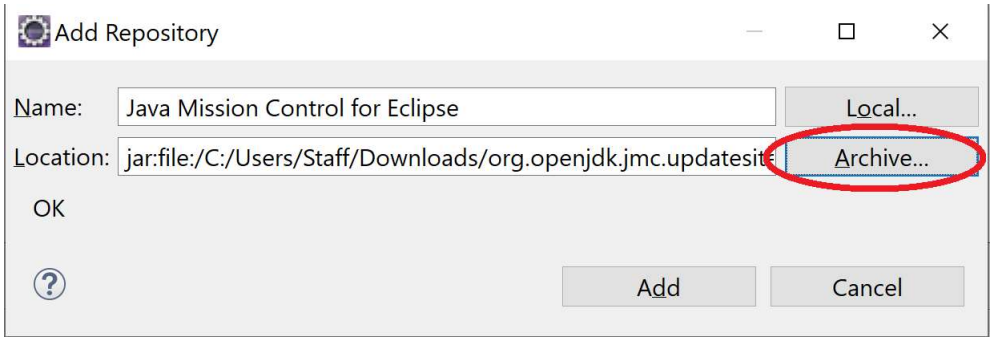
Under the Eclipse 'Help' menu, select 'Install New Software…':



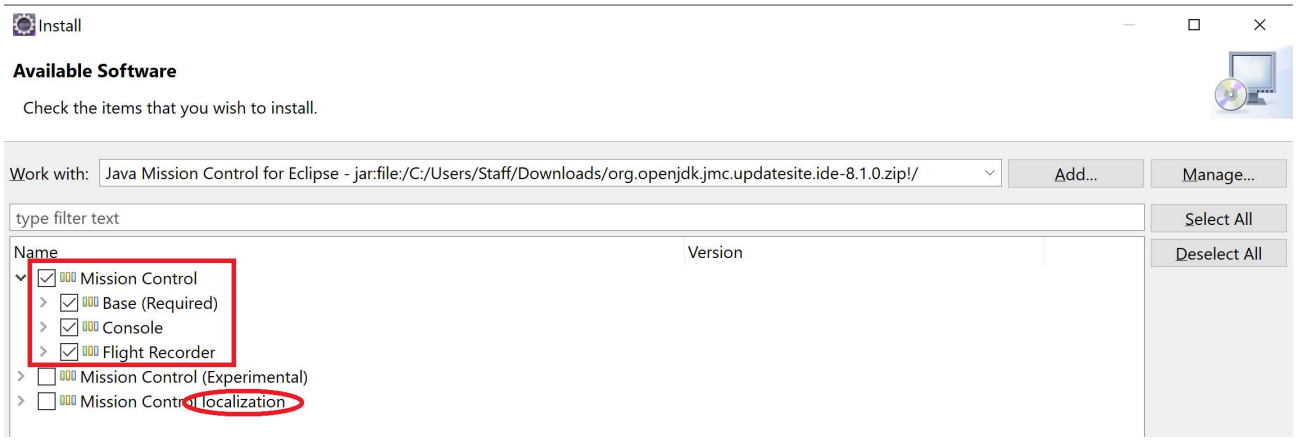
Click the ‘Add’ button:



In the ‘Name’ field you can type anything you wish, for example ‘Java Mission Control for Eclipse’. For the ‘Location’ field, click the Archive button and locate the archive file you have downloaded and saved in the previous step:



After clicking ‘Add’, you will be given the choice to install Mission Control (Note: Ignore the Experimental.). You may install the ‘localization’ module if you wish to have Chinese and Japanese support:



Click the ‘Next’ button, accept the license and then ‘Finish’ – Eclipse will need a restart.

Note that if you cannot install the Java Mission Control as a plugin inside of your Eclipse, you

may install and run it as a separate application. You can download the Java Mission Control installer from https://www.oracle.com/java/technologies/javase/products-jmc8-downloads.html .

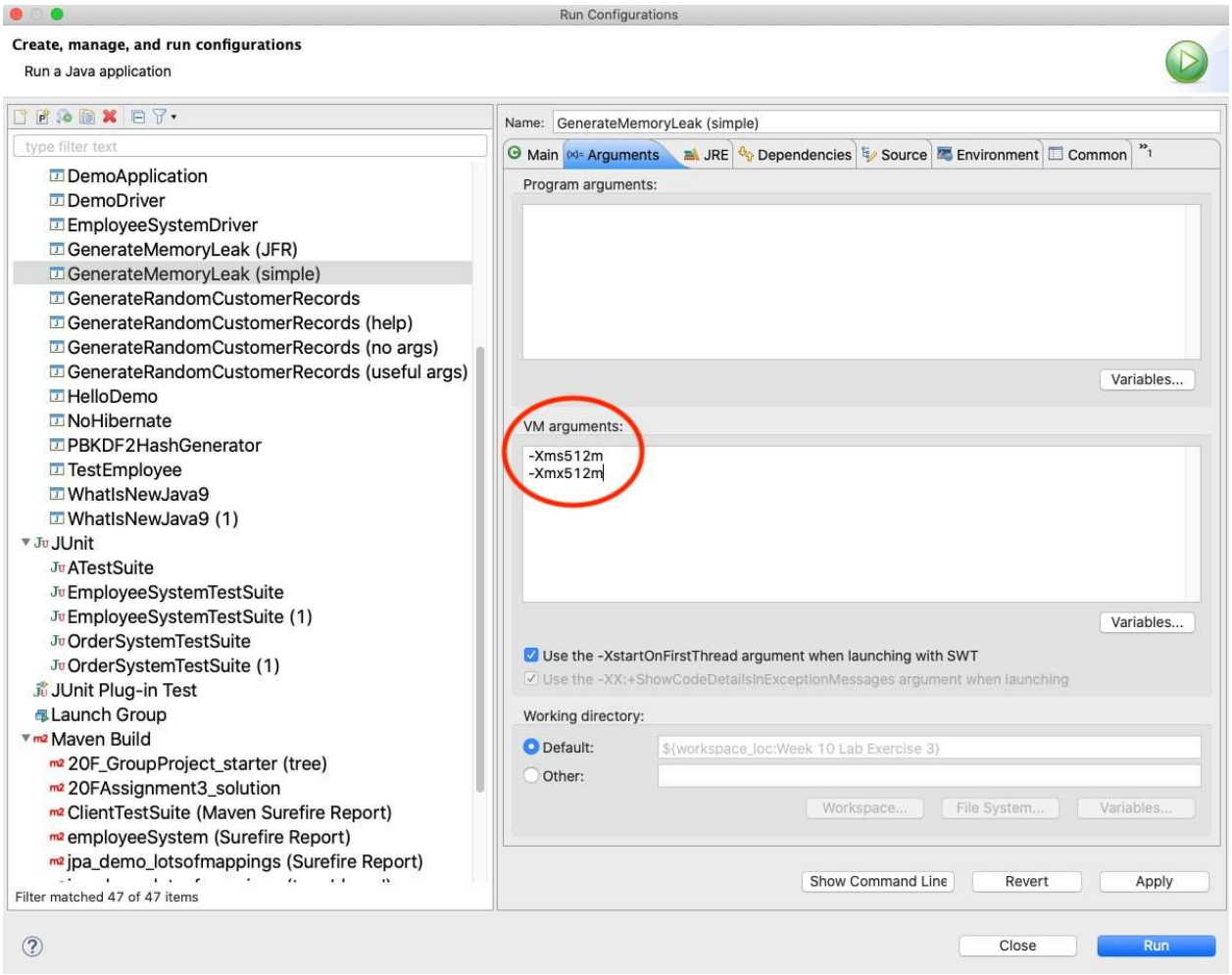
Installing the MemoryLeak Application

In Brightspace under Activities->Assignments, you will find the 'Lab 03 – Java Mission Control'. Attached is 'MemoryLeak.zip'. Please import this application to your Eclipse as an Existing Maven Project.

When you run the GenerateMemoryLeak main program, it will – depending on your machine – take somewhere between 1 to 5 minutes to crash with the following exception:

```
Exception in thread "Producer Thread" java.lang.OutOfMemoryError: Java heap  space
```

To make this happen faster, we can reduce the amount of RAM given to the JVM when the program runs (you will have to run your program once before getting access to Arguments):



Now it should not take much more than 30 seconds to trigger the problem.

Note: Even when the OutOfMemoryError happens, the program is still actually running – you will need to stop it (red button in the Eclipse console).

Profiling the MemoryLeak Application

In order for JFR to provide any analysis, we need its output – JFR is enabled using a JVM arg:

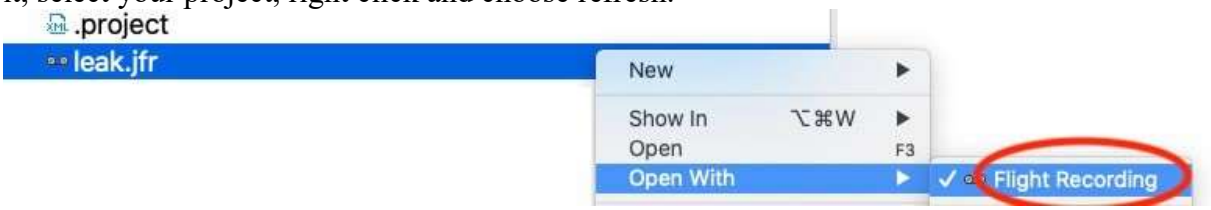
```
-Xms512m
-Xmx512m
-Dcom.sun.management.jmxremote
-XX:StartFlightRecording=duration=30s,settings=profile,filename=leak.jfr
```

JFR settings:

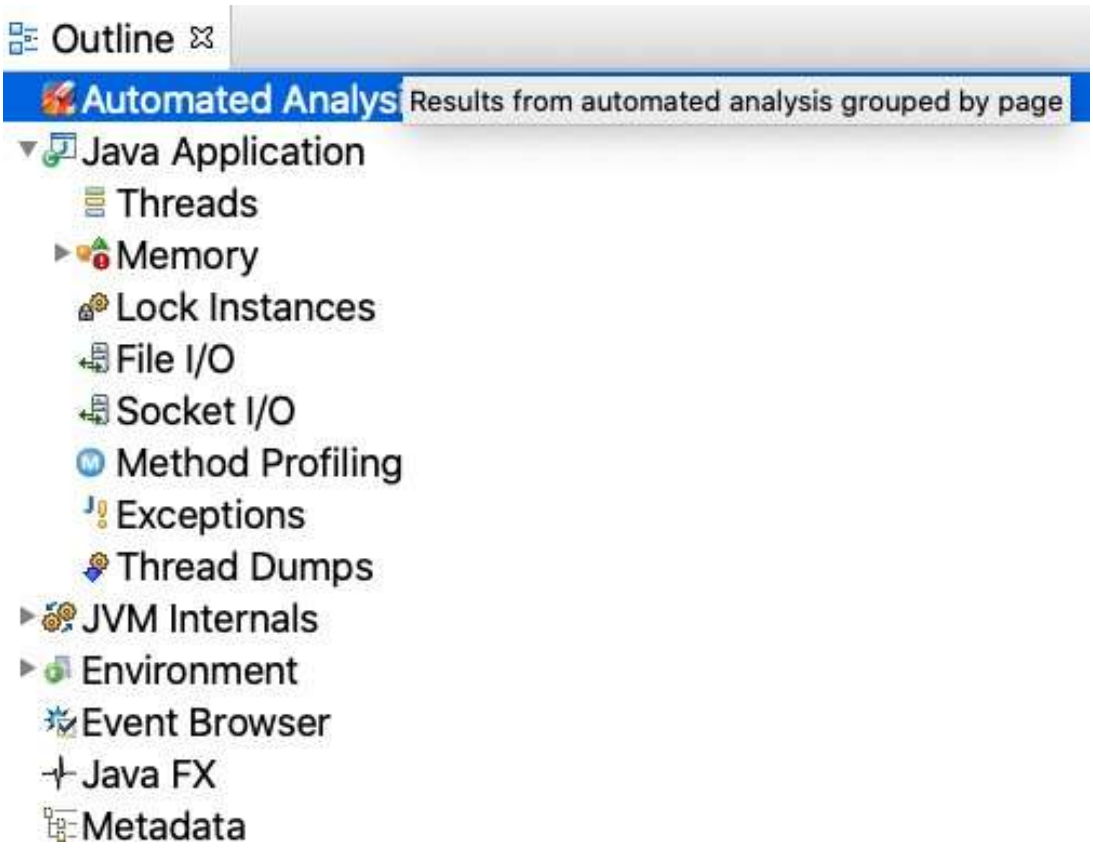
- Duration: length of time to record profiling data; if not specified, recording will continue indefinitely – in practice, typically one would send an explicit JFR.Stop command.
- Settings: the type of profiling data collected:
  - default (default when not specified): a pre-determined set of profiling data is collected. Collecting profiling data is not without some impact on the program – this profile collects information with a very low overhead on the program's performance (1 to 2%) and is typically used with recordings that run for a long time
  - profile: collects more data than the default profile with more overhead and thus can have an impact on the program's normal operation. Typically, this configuration is used for shorter durations when more information is needed
  - custom: it is possible to register a custom profile using the 'Flight Recorder Template Manager' which allow one to target specific data to be collected and change thresholds or timeouts (advanced)
- Filename: name of the file containing the profiling data. If the name does not contain any path information, the file will be in the directory where the process was started. Typically, the filename extension is '.jfr' but that is not enforced.

Analysis

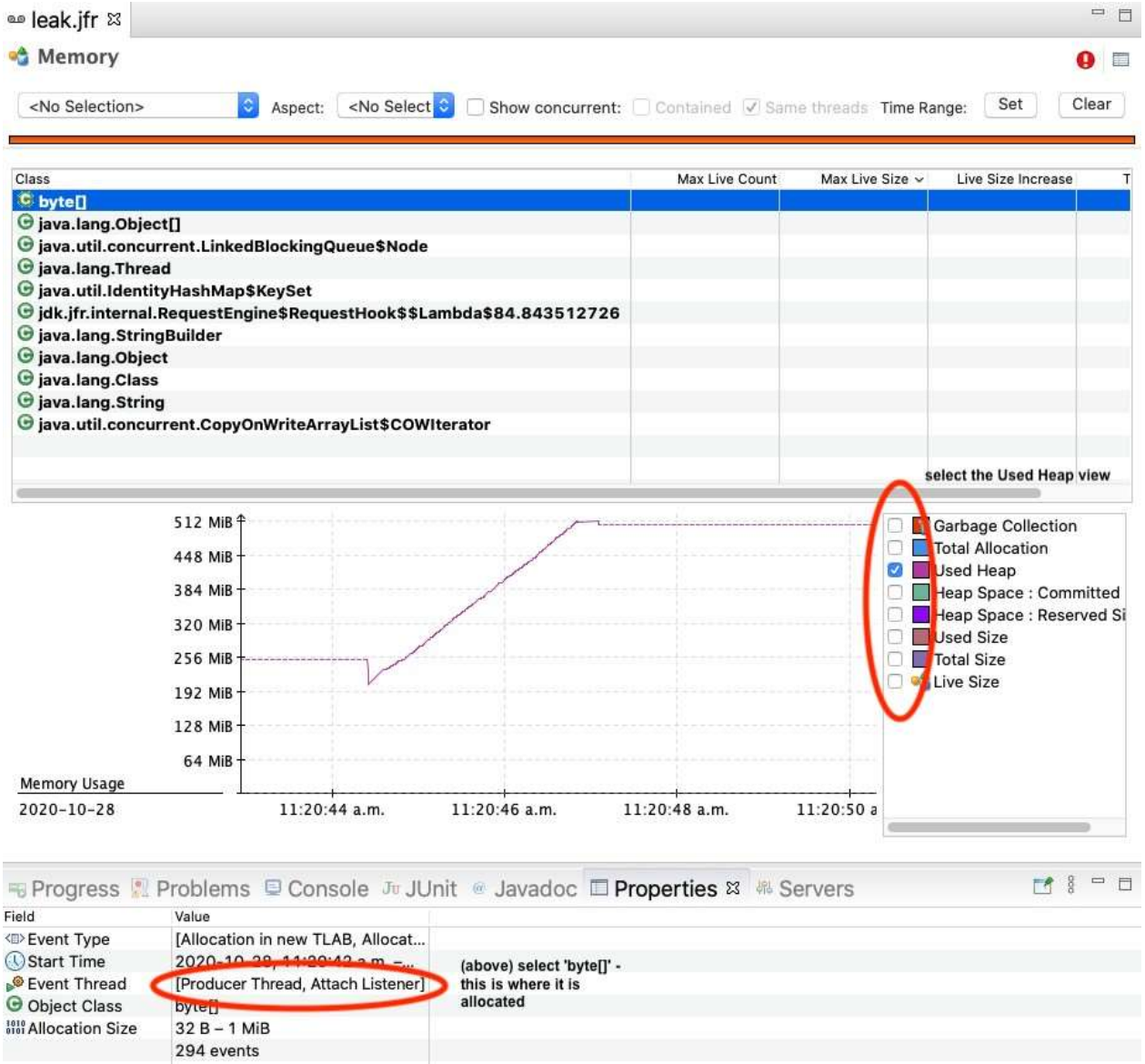
Open the leak.jfr file. You will have to wait the duration (30s default) defined in the arguments of your run. This file is in the main directory of your project. If you cannot see it, select your project, right click and choose refresh.



You need to find the Eclipse ‘Outline’ view – it shows a variety of ways to visualize the data:



If you select the ‘Memory’ view, you can see several interesting things:



Toggle the Memory Usage buttons so that only ‘Used Heap’ is selected – now we can see the steady increase in memory up to the max (remember the JVM args we added, 512M).

In the ‘Class’ view, select **byte[]** – find the Eclipse ‘Properties’ view and you can see it says objects of this type are allocated in the Producer Thread. We can verify this in GenerateMemoryLeak, line 23:

```
            queue.offer(new byte[1 * 1024 * 1024]);
```

Submission

You will need to run both of provided classes in your skeleton zip. One has memory leak, and the other does not.

Arguments for the one with memory leak is below. If OutOfMemoryError does not happen within 30 seconds just increase the duration by 10 second increments:

```
-Xms512m
-Xmx512m
-Dcom.sun.management.jmxremote
-XX:StartFlightRecording=duration=30s,settings=profile,filename=leak.jfr
```

Arguments for the one without memory leak:

```
-Xms512m
-Xmx512m
-Dcom.sun.management.jmxremote
-XX:StartFlightRecording=duration=30s,settings=profile,filename=noleak.jfr
```

In both of the Java code files, you will find:
new Thread(producer, ***"Producer Thread"***).start();

Replace the "Producer Thread" with your name and student number.
Ex. "Teddy-Yap-0123456"

Finally, like in page 7, take a screenshot of your IDE with the same visible components and your name visible in properties. Do this for both Java codes. You don't need to circle or crop anything, as long byte[], Used Heap, and Event Thread in Properties tab are clearly visible.

Submit both of your screenshots to Lab 3 on Brightspace. There is no need to demo anything for this lab.

- End –